POWER PROGRAMMING

Writing Portable Programs For Windows

BY RAY DUNCAN

In the last column, I introduced you to the so-called WIN16, WIN32S, and WIN32 APIs. As you'll recall, we're using the term WIN16 for the 16-bit API of Windows 3.1, WIN32 for the expanded 32-bit API that was designed by Microsoft to compete with OS/2 2.0's 32-bit Presentation Manager API, and WIN32S for that subset of the WIN32 API that has direct counterparts in WIN16.

At the present time, WIN32 is supported only on Microsoft's new NT operating system for 386 and 486 processors and MIPS machines; a full-fledged WIN32 implementation that runs on top of DOS has been promised for 1993 (though I'm skeptical that it will ever materialize). In the meantime, Microsoft is supporting WIN32S on DOS and Windows 3.*x* with a special loader in a VxD and a "thunk" layer that translates 32-bit API calls by the application into 16-bit API calls. The exact same WIN32S binaries run on either Windows 3.*x* or NT/WIN32. As a temporizing solution, this is quite reasonable, and it should motivate many developers to move their applications to 32-bit mode much earlier than they would otherwise.

NT has many powerful and interesting features, but we're going to ignore them for a while yet and talk instead about the issue of source code portability between WIN16, WIN32S, and WIN32. Since WIN32S and WIN32 are supersets of WIN16, and WIN32S is almost exactly symmetric with WIN16, it's quite feasible to compile programs for either the WIN16 or WIN32 execution environment from the same source code base. The flat memory model of WIN32, the widening of integers from 16 bits to 32 bits, and the differences in calling conventions are generally transparent, if sufficient attention is paid to the correct use of data types and handles. The other inescapable differences between the 16-bit and 32-bit programming interfaces are summarized in Figure 1.

Don't allow yourself to be intimidated by Figure 1; it may look like a long list of things to fret about, but in fact the items listed have surprisingly little impact on application source code. The "vanished" functions are principally those that are CPU-architecture-dependent, and they must be dealt with on an individual basis. On the other hand, the semantics of the altered APIs and altered messages usually haven't changed, so most of the differences in that area can be handled by mechanical editing and conditional #defines in the application header file. For example, the changes to messages are almost always related to parameter packing—typically the movement of some value from lParam to wParam to allow for a 32-bit handle in lParam. The superseded functions are mostly sound functions that have been replaced by the multimedia API, or by relatively esoteric GDI functions related to coordinate spaces.

Our sample program, a utility for viewing a file in hex and binary, demonstrates the surprisingly easy portability between the 16- and 32-bit Windows environments.

To help convince you that this easy portability from WIN16 to WIN32 isn't just standard Microsoft marketing hype but is in fact achievable, I'm going to follow up last issue's WINAPP application skeleton with a considerably more elaborate programming example. HEXVIEW, which can be seen in action in Figure 2, is a relatively mundane utility for viewing the contents of a file in hex and binary. Looking at the utility is worthwhile, though, because it demonstrates many of the aspects of a functional Windows application: It has a menu and an About dialog, it exploits the common dialog library, it performs file I/O, it selects a font, it paints text in its window, it maintains a scroll bar, and it saves and retrieves preference information about the size and location of its window and the name of the most recent file viewed. (Just to keep the full picture in perspective, the important Windows application tasks that HEXVIEW *doesn't* demonstrate are keyboard input, Clipboard operations, drag-and-drop, and DDE.) You can download HEXVIEW from PC MagNet, archived as HEXVIE.ZIP.

Pay attention now as we list all the ways that the HEXVIEW source code can be compiled and executed (see Figure 3) without any changes whatsoever! If HEXVIEW is compiled with BC++ 3.0, we get a 16-bit protected-mode NE binary that can run under Windows 3.*x* on 286, 386, or 486 processors; under NT on 386 or 486 processors using its Windows-On-Windows compatibility layer (WOW); or under NT on RISC processors using emulation technology that Microsoft has licensed from Insignia Corp. If HEXVIEW is compiled with Microsoft's CL386, we get a 32-bit protected-mode PE binary that can be run

the WIN32S translation DLLs and VxD, or under NT on a 386 or 486 as a native application. If HEXVIEW is compiled with the MIPS NT compiler, we get a 32-bit MIPS binary that can be run under NT on the R-4000 processor.

SOURCE CODE OVERVIEW The source code for HEXVIEW is listed in Figure 4, the header file in Figure 5, the resource script in Figure 6, and the module definition file in Figure 7. The header file consists mostly of function prototypes, except for a few arbitrary menu identifiers and some conditional #defines that we'll return to later. The resource script contains a totally standard, totally familiar Windows menu definition, dialog template, and icon declaration; we won't discuss their arcane syntax further here, except to remind you that it's the same for all of the target environments.

Turning to the source file, let's sort out the routines by their function and when they're executed, then look at the intent of each procedure in a little more detail.

- Application Infrastructure: WinMain, InitApp, InitInstance, TermInstance.
- Event Handling for the Frame Window: FrameWndProc, DoDestroy, DoClose, DoVScroll, ThumbTrack, DoSize, DoPaint, DoCommand, DoMenuOpen, DoMenuAbout, DoMenuExit.
- Event Handling for the Child Window: ChildWndProc, DoChildPaint.
- Event Handling for the About Dialog Box: AboutDlgProc.
- File Manipulation: OpenDataFile, ReadDataFile, GetByte.
- Helper Routines: DisplayLine, ConfigDisplay, Repaint, SetWindowCaption, SetFilePosition, UpdateFrameProfile.

It's a bit ironic that as a C program becomes more modular and structured, the procedures within the program become shorter and more numerous, and the source code correspondingly more difficult to understand without a road map or code-analysis tools. Good structured technique and modularity are, after all, supposed to improve the maintainability of a program, and so it does, but only after the usual learning curve. Newcomers to Windows programming also find it strange that a Windows program contains routines that are apparently never called from the main sequence of execution; only after they understand oriented architecture does the source code begin to make some kind of sense.

INFRASTRUCTURE PROCEDURES The procedures WinMain, InitApp, InitInstance, and TermInstance—along with the C runtime library startup code, which is linked into the program automatically and is invisible to us in this source listing—can be thought of as the program's infrastructure. When HEXVIEW is launched, the startup code first receives control, and after performing some housekeeping and several calls to Windows API functions, it transfers control to the WinMain routine.

WinMain calls or does not call InitApp according to the value of hInstance, one of its parameters. Under Windows 3.*x*, if hInstance is zero, this instance of HEXVIEW is the only one running, and InitApp must be called to register the application's window classes. If hInstance is nonzero under Windows 3.*x*, another instance is already running, and InitApp is bypassed. hInstance is always zero in a WIN32 or WIN32S environment, so InitApp will always be called when HEXVIEW is compiled as a 32-bit application.

Following the possible call to InitApp, WinMain calls InitInstance to create the application's frame window, size and position the window according to saved preferences from a previous execution of the program, get a handle for a nonproportional font, allocate some memory for a file I/O buffer, and reopen the file that was being viewed when the program was last executed. WinMain then falls into the famous (or infamous) Windows event message queue and dispatching them t the appropriate window callback routine. When a WM_QUIT message is received, the "value" of GetMessage is zero and WinMain falls out of the loop, calls TermInstance to close the file and release the memory allocation, and then exits back to Windows. Wait a minute! Now that we've traced through WinMain from beginning to end and looked at every procedure that it calls, we still haven't seen where the real work of the program is accomplished! That's because the most important logic of the program is actually outside the main line of execution, in the event-handling routines.

EVENT-HANDLING PROCEDURES The front line of event handling for HEXVIEW is the routine FrameWndProc. Whenever the user moves or clicks the mouse on any part of HEXVIEW's window or hits a key while HEXVIEW has the input focus, or when any other event occurs that Windows thinks might be of interest to HEXVIEW, a message packet is placed in HEXVIEW's task queue. WinMain dequeues the message by calling GetMessage and passes the messag to FrameWndProc by calling Dispatch Message. (FrameWndProc was designated as the message handler for HEXVIEW's frame window as part of the process of registering the window class in InitApp. DispatchMessage looks at the window handle in the message, looks up the appropriate message handler by backtracking from the window handle to the window class of which it is a member, and then "calls back" into HEXVIEW

Differences Between the WIN16 and WIN32 APIs

1 of 2

WIN16 Functions Not Supported in WIN32	WIN16 Functions Altered or Superseded in WIN32	WIN16 Messages Altered or Superseded in WIN32
AccessResource	AddFontResource	EM_GETSEL
AllocDStoCSAlias	CloseComm	EM_LINESCROLL
AllocResource	CloseSound	EM_SETSEL
AllocSelector	CountVoiceNotes	WM_ACTIVATE
ChangeSelector	DeviceCapabilities	WM_CHANGECBCHAIN
Dos3Call	DeviceMode	WM_CHARTOITEM
FreeSelector	DlgDirSelect	WM_COMMAND
GetCodeHandle	DlgDirSelectComboBox	WM_CTLCOLOR
GetCodeInfo	ExtDeviceMode	WM_DDE_ACK
GetCurrentPDB	FlushComm	WM_DDE_ADVISE
GetEnvironment	GetAspectRatioFilter	WM_DDE_DATA

Figure 1: **Summary of the inescapable differences between the 16-bit and 32-bit programming interfaces. The functions in WIN32 that have no counterparts in WIN16 are not shown here.**

Differences Between the WIN16 and WIN32 APIs

2 of 2

WIN16 Functions Not Supported in WIN32	WIN16 Functions Altered or Superseded in WIN32	WIN16 Messages Altered or Superseded in WIN32
GetInstanceData	GetBitmapDimension	WM_DDE_EXECUTE
GetKBCodePage	GetBrushOrg	WM_DDE_POKE
GetTempDrive	GetClassWord	WM_HSCROLL
GlobalDosAlloc	GetCommError	WM_MDIACTIVATE
GlobalDosFree	GetCurrentPosition	WM_MDISETMENU
GlobalPageLock	GetMetaFileBits	WM_MENUCHAR
GlobalPageUnlock	GetModuleUsage	WM_MENUSELECT
LimitEMSPages	GetTextExtent	WM_PARENTNOTIFY
LocalNotify	GetTextExtentEx	WM_VKEYTOITEM
NetBIOSCall	GetThresholdEvent	WM_VSCROLL
ProfClear	GetThresholdStatus	
ProfFinish	GetViewportExt	
ProfFlush	GetViewportOrg	
ProfInsChk	GetWindowExt	
ProfSampRate	GetWindowOrg	
ProfSetup	GetWindowWord	
ProfStart	MoveTo	
ProfStop	OffsetViewportOrg	
SetEnvironment	OffsetWindowOrg	
SetResourceHandler	OpenComm	
SwitchStackBack	OpenSound	
SwitchStackTo	ReadComm	
UngetCommchar	RemoveFontResource	
ValidateCodeSegments	ScaleViewportEx	
ValidateFreeSpaces	ScaleWindowEx	
	SetBitmapDimension	
	SetClassWord	
	SetCommMask	
	SetMetaFileBits	
	SetSoundNoise	
	SetViewportExt	
	SetViewportOrg	
	SetSoundAccent	
	SetVoiceEnvelope	
	SetVoiceNote	
	SetVoiceQueueSize	
	SetVoiceSound	
	SetVoiceThreshold	
	SetWindowExt	
	SetWindowOrg	
	SetWindowWord	
	StartSound	
	StopSound	
	SyncAllVoices	
	WaitSoundState	
	WriteComm	

at the message handler's entry point.)

When FrameWndProc is entered, the various components of the message currently being processed are on the stack: hWnd (the window handle), wMsg (an integer value that indicates the message type), and two parameters called wParam and lParam whose contents and significance depend on the message type. FrameWndProc looks up the message type in the table frameMsgs[] and, if it finds a match, passes the message to the appropriate routine. If the message type isn't found in the table frameMsgs[], FrameWndProc passes it to the Windows API function DefWindowProc, which contains code for the default behavior for each message type.

The HEXVIEW procedures that are called by FrameWndProc are mostly small and straightforward. DoClose processes a WM_CLOSE message by calling UpdateFrameProfile to save the window's current size and position and the current filename and file pointer, then forcing a WM_DESTROY message by calling the Windows function DestroyWindow. DoDestroy processes the WM_DESTROY message by forcing a WM_QUIT message (which causes WinMain to fall out of its event loop). DoSize reacts to a window-resizing operation by calculating the number of lines that will now fit into the window, updating certain global variables, and calling ConfigWindow to configure the number of output lines that correspond to each scroll-bar unit and update the scroll bar on the screen accordingly.

The procedure DoCommand is called when FrameWndProc receives any sort of message from the menu bar, triggered by the user's mouse click on the menu bar or by accelerator keys for menu items. DoCommand decodes the message with the aid of the table menuitems[], then hands off control to DoMenuOpen, DoMenuExit, or DoMenuAbout to open a file, terminate the application, or display the About dialog box, respectively. DoMenuOpen calls the common dialog library routine GetOpenFileName to get a filename and uses the routines OpenDataFile, ReadDataFile, and Repaint to open the file, initialize the file I/O buffer and some global variables, and refresh the display in the main window. DoMenuExit responds by sending a WM_CLOSE message to the frame window, exactly as though the user had picked Close from the system menu, allowing shutdown processing to be unified in the routine DoClose. DoMenuAbout creates and displays the About dialog box, using the static dialog resource built into the executable file (by the resource compiler from the About dialog template in HEXVIEW.RC). Once the dialog is active, the callback routine AboutDlgProc grabs control until the dialog is dis-

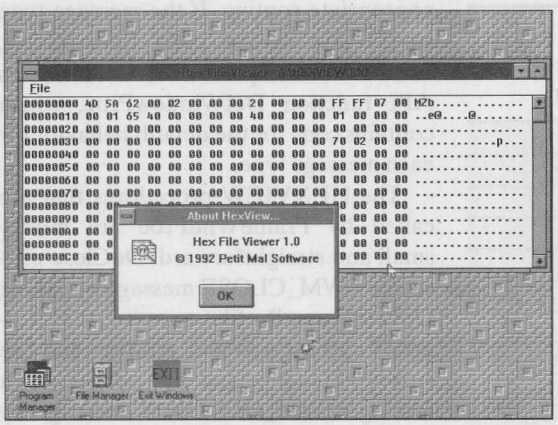

***Figure 2:* The HEXVIEW program in action.**

missed by clicking the OK button.

The DoPaint routine is entered when a WM_PAINT message indicates that some part of the window needs updating. This message might be received because the window was moved, resized, iconized and then restored, or covered up and then revealed, or even because HEXVIEW intentionally sent a message to itself (in the routine Repaint) when a new file was opened or the file pointer was moved by the scroll-bar message–processing routine. DoPaint obtains a device context (DC), selects a nonproportional font into the device context using the font handle that was originally obtained by InitInstance, and then calls the helper routine DisplayLine to format and display each line of output within the window. DisplayLine uses the helper routine GetByte to extract each binary data byte from the file I/O buffer for conversion and formatting; any file I/O that is needed is hidden within GetByte.

DoVScroll is probably the most interesting message-handling routine in HEXVIEW. It is called whenever there is a vertical scroll-bar event—whenever the user clicks on the scroll bar, clicks on the arrows at the ends of the scroll bar, or drags the "thumb." DoVScroll decodes the event type, calculates the new file position with the aid of the helper routine SetFilePosition, and updates the display accordingly (usually by calling Repaint to force a WM_PAINT message). There's more code in DoVScroll than you'd expect, mainly because we want to minimize the amount of time spent painting in the window. DoVScroll checks for cases where there is nothing to do or where it can BitBlt the current contents of the window up or down by one line (using the Windows API function ScrollWindow), and then it lets DoPaint paint the "missing" line.

When the user drags the thumb, DoVScroll calls ThumbTrack to create a tiny child window in the middle of the main window, calculate the file offset that corresponds to the new thumb position, and send WM_PAINT messages to the child window's message handler, ChildWndProc. ChildWndProc decodes its message using the table childMsgs[] and calls DoChildPaint for WM_PAINT messages to display the file offset as feedback for the user. When the user releases the thumb, DoVScroll calculates the new file position and updates the main window. The destruction of the child window happens automatically in the helper routine SetFilePosition.

FILE MANIPULATION PROCEDURES We've already discussed DoMenuOpen, which is called by FrameWndProc in response to a File Open menu command by the user. DoMenuOpen uses the common dialog library routine GetOpenFileName to retrieve a valid filename from the user, then calls OpenDataFile to open the file and fills the file I/O buffer by calling ReadDataFile. OpenDataFile initializes certain global variables including the file handle and current file position, reconfigures the scroll bar with the aid of ConfigDisplay, and puts the filename into the title bar by calling SetWindowCaption. OpenDataFile and ReadDataFile may also be called from InitInstance if a filename was saved by a previous execution of HEXVIEW.

From the standpoint of display logic (DoPaint and its helpers), the entire file is always accessible in memory, and any given byte of the file can be retrieved b calling GetByte with a file offset. GetByte knows how much data and which part of the file is currently in the file I/O buffer and calls ReadDataFile whenever necessary to alter the contents of the buffer, implementing a sort of poor-man's virtual-memory scheme. SetFilePosition is called from many routines in the program but principally in response to scroll-bar events; it alters the global variables that indicate the current file position for display in the window and updates the scroll-bar thumb position accordingly, but does no file I/O. A call to Repaint, which typically follows SetFilePosition, causes DoPaint to execute via a forced WM_PAINT message; DoPaint's calls to GetByte will cause file I/O to occur if it is needed.

HELPER ROUTINES All of the helper routines are called indirectly by Frame WndProc as a result of message processing, and a few are also called by InitInstance to set up the display of a file that was open when HEXVIEW was last executed. Each of the helper routines has already been discussed in the context o event handling and file I/O procedures.

PORTABILITY CONSIDERATIONS Now that we've had a whirlwind tour through HEXVIEW, let's go back and see where portability considerations intrude into the source code.

First, we must remember that the general strategy in creating the WIN32 API from the WIN16 API was to widen all 16-bit parameters to 32 bits and to replace 32-bit segment:offset pointers with 32-bit flat-model near offsets. This aspect can be handled fairly completely by treating FAR as a NOOP for 32-bit code and by hiding the real data types behind TYPEDEFs such as HWND, BOOL, INT, UINT, and LONG. If you look at the

Execution Environments for HEXVIEW

Compiled with	Windows 3.*x* on a 286 PC	Windows 3.*x* on a 386 or 486 PC	NT on a 386 or 486 PC	NT on a MIPS R-4000
BC++ 3.0	Yes (native)	Yes (as 16-bit app)	Yes (using WOW interface layer)	Yes (using emulation technology)
MS CL386	No	Yes (using WIN32S DLLs and VxD)	Yes (native)	No
MIPS CC	No	No	No	Yes (native)

***Figure 3:* These are the various ways the HEXVIEW source code can be compiled and executed.**

```
// HexView - Windows 3.x and NT/WIN32 Hex File Viewer
// Copyright (C) 1992 Ray Duncan
// PC Magazine * Ziff Davis Publications

#define dim(x) (sizeof(x) / sizeof(x[0]))   // returns no. of elements
#define EXENAMESIZE 256                     // max length of path+filename
#define BUFSIZE 65520                       // size of file I/O buffer
#define BPL 16                              // bytes per line

#include "stdlib.h"
#include "windows.h"
#include "string.h"
#include "commdlg.h"
#include "hexview.h"

HANDLE hInst;                               // module instance handle
HWND hFrame;                                // handle for frame window
HWND hChild;                                // handle for child window
HFONT hFont;                                // handle for nonprop. font
INT CharX, CharY;                           // character dimensions
INT LPP;                                    // lines per page
INT BPP;                                    // bytes per page
INT ThumbInc;                               // bytes per thumb unit
INT hFile = -1;                             // handle for current file
char szFileName[EXENAMESIZE+1];             // name of current file
HANDLE hBuff;                               // handle for file I/O buffer
LPSTR lpBuff;                               // far pointer to file buffer
LONG ViewPtr;                               // addr, first line current page
LONG FilePtr;                               // addr, start of i/o buffer
LONG FileSize;                              // length of current file
LONG FileIndex;                             // index from thumb tracking
LONG TopAddr;                               // address, top of last page
LONG TopLine;                               // line num, top of last page
INT WinWidth;                               // width of frame client area

char szFrameClass[] = "HexView";            // classname for frame window
char szChildClass[] = "HexViewChild";       // classname for child window
char szAppName[] = "Hex File Viewer";       // long application name
char szMenuName[] = "HexViewMenu";          // name of menu resource
char szIconName[] = "HexViewIcon";          // name of icon resource
char szIni[] = "Hexview.Ini" ;              // initialization filename

char *szFilter[] = {                        // filters for Open dialog
    "All Files (*.*)", "*.*",
    "Executable files (*.EXE)", "*.EXE",
    "Object Modules (*.OBJ)", "*.OBJ",
    "" };

//
// Table of window messages supported by FrameWndProc()
// and the functions which correspond to each message.
//
struct decodeWord frameMsgs[] = {
    WM_PAINT, DoPaint,
    WM_SIZE, DoSize,
    WM_COMMAND, DoCommand,
    WM_CLOSE, DoClose,
    WM_DESTROY, DoDestroy,
    WM_VSCROLL, DoVScroll, } ;

//
// Table of window messages supported by ChildWndProc()
// and the functions which correspond to each message.
//
struct decodeWord childMsgs[] = {
    WM_PAINT, DoChildPaint, } ;

//
// Table of menubar item IDs and their corresponding functions.
//
struct decodeWord menuitems[] = {
    IDM_OPEN, DoMenuOpen,
    IDM_EXIT, DoMenuExit,
    IDM_ABOUT, DoMenuAbout, } ;

//
// WinMain -- entry point for this application from Windows.
//
INT APIENTRY WinMain(HANDLE hInstance,
    HANDLE hPrevInstance, LPSTR lpCmdLine, INT nCmdShow)
{
    MSG msg;                                // receives message packet

    hInst = hInstance;                      // save instance handle

    if(!hPrevInstance)                      // if first instance,
        if(!InitApp(hInstance))             // register window class
        {
            MessageBox(hFrame, "Can't initialize HexView!", szAppName,
                MB_ICONSTOP | MB_OK);
            return(FALSE);
        }

    if(!InitInstance(hInstance, nCmdShow))  // create instance window
    {
        MessageBox(hFrame, "Can't initialize HexView!", szAppName,
            MB_ICONSTOP | MB_OK);
        return(FALSE);
    }

    while(GetMessage(&msg, NULL, 0, 0))     // while message != WM_QUIT
    {
        TranslateMessage(&msg);             // translate virtual key codes
        DispatchMessage(&msg);              // dispatch message to window
    }

    TermInstance(hInstance);                // clean up for this instance
    return(msg.wParam);                     // return code = WM_QUIT value
}

//
// InitApp --- global initialization code for this application.
// Registers window classes for frame and child windows.
//
BOOL InitApp(HANDLE hInstance)
{
    WNDCLASS  wc;                           // window class data
    BOOL bParent, bChild;                   // status from RegisterClass

    // set parameters for frame window class
    wc.style = CS_HREDRAW|CS_VREDRAW;       // class style
    wc.lpfnWndProc = FrameWndProc;          // class callback function
    wc.cbClsExtra = 0;                      // extra per-class data
    wc.cbWndExtra = 0;                      // extra per-window data
    wc.hInstance = hInstance;               // handle of class owner
    wc.hIcon = LoadIcon(hInst, "HexViewIcon");      // application icon
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);       // default cursor
    wc.hbrBackground = GetStockObject(WHITE_BRUSH); // background color
    wc.lpszMenuName =  szMenuName;          // name of menu resource
    wc.lpszClassName = szFrameClass;        // name of window class

    bParent = RegisterClass(&wc);           // register parent window class

    // modify some of the parameters for the child window class
    wc.style = 0;                           // class style
    wc.lpfnWndProc = ChildWndProc;          // class callback function
    wc.lpszMenuName =  NULL;                // name of menu resource
    wc.lpszClassName = szChildClass;        // name of window class

    bChild = RegisterClass(&wc);            // register child window class

    return(bChild && bParent);              // return combined status
}

//
// InitInstance --- instance initialization code for this application.
// Gets information about system nonproportional font.  Allocates memory
// to use as file I/O buffer.  Creates frame window, gets initialization
// information (if any) from INI file (Windows 3.x) or registration
// database (WIN32/NT). Positions and sizes window, opens and positions
// file if initialization info found.
//
BOOL InitInstance(HANDLE hInstance, INT nCmdShow)
{
    HDC hdc;                                // handle for device context
    TEXTMETRIC tm;                          // font information
    RECT rect;                              // window position & size
    char buff[80];                          // scratch buffer

    hFrame = CreateWindow(                  // create frame window
        szFrameClass,                       // window class name
        szAppName,                          // text for title bar
        WS_OVERLAPPEDWINDOW | WS_VSCROLL,   // window style
        CW_USEDEFAULT, CW_USEDEFAULT,       // default position
        CW_USEDEFAULT, CW_USEDEFAULT,       // default size
        NULL,                               // no parent window
        NULL,                               // use class default menu
        hInstance,                          // window owner
        NULL);                              // unused pointer

    if(!hFrame) return(FALSE);              // error, can't create window

    hdc = GetDC(hFrame);                    // get device context
    hFont = GetStockObject(SYSTEM_FIXED_FONT);
    SelectObject(hdc, hFont);               // realize nonproportional
    GetTextMetrics(hdc, &tm);               // font, get character size,
    CharX = tm.tmAveCharWidth;              // and calculate window width
    CharY = tm.tmHeight + tm.tmExternalLeading;
    WinWidth = (CharX * 75) + GetSystemMetrics(SM_CXVSCROLL);
    ReleaseDC(hFrame, hdc);                 // release device context

    if(!(hBuff = GlobalAlloc(GMEM_MOVEABLE, BUFSIZE)))  // allocate memory
        return(FALSE);                      // error, out of memory

    lpBuff = GlobalLock(hBuff);             // get far pointer to memory

    GetWindowRect(hFrame, &rect);           // get window position, size

    // read saved position/size profile for our frame window, if any
    rect.left = GetPrivateProfileInt("Frame", "xul", rect.left, szIni);
    rect.top = GetPrivateProfileInt("Frame", "yul", rect.top, szIni);
    rect.right = GetPrivateProfileInt("Frame", "xlr", rect.right, szIni);
    rect.bottom = GetPrivateProfileInt("Frame", "ylr", rect.bottom, szIni);

    MoveWindow(hFrame, rect.left, rect.top, // force window position, size
        WinWidth, rect.bottom-rect.top, TRUE);
```

Figure 4: **This is the C-language source code for HEXVIEW, a utility for viewing the contents of a file in hex and binary form.**

HEXVIEW.C

2 of 4

```
    // get saved filename and file offset for display, if any
    GetPrivateProfileString("File", "filename", "", szFileName,
        EXENAMESIZE, szIni);
    GetPrivateProfileString("File","fileptr","",buff, sizeof(buff),szIni);

    if(szFileName[0])                       // if filename and file offset
    {                                       // was saved from previous
        OpenDataFile();                     // execution, open the file
        ReadDataFile();                     // and set file position
        SetFilePosition(atol(buff));
    }

    ShowWindow(hFrame, nCmdShow);           // make frame window visible
    UpdateWindow(hFrame);                   // force WM_PAINT message
    return(TRUE);                           // return success flag
}

//
// TermInstance -- instance termination code for this application.
// This is a general-purpose opportunity to clean up after ourselves.
//
BOOL TermInstance(HANDLE hinstance)
{
    if(hFile != -1)                         // close file if any
        _lclose(hFile);

    GlobalUnlock(hBuff);                    // unlock the memory buffer
    GlobalFree(hBuff);                      // release the buffer
    return(TRUE);                           // return success flag
}

//
// FrameWndProc --- callback function for application frame window.
// Decodes window message using table frameMsgs[] and runs corresponding
// function.  If no match found, passes message to Windows DefWindowProc().
//
LONG FAR APIENTRY FrameWndProc(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    INT i;

    for(i = 0; i < dim(frameMsgs); i++)
    {
        if(wMsg == frameMsgs[i].Code)
            return((*frameMsgs[i].Fxn)(hWnd, wMsg, wParam, lParam));
    }

    return(DefWindowProc(hWnd, wMsg, wParam, lParam));
}

//
// ChildWndProc --- callback function for application child window.
// Works like FrameWndProc, except uses table childMsgs[].
//
LONG FAR APIENTRY ChildWndProc(
HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    INT i;

    for(i = 0; i < dim(childMsgs); i++)
    {
        if(wMsg == childMsgs[i].Code)
            return((*childMsgs[i].Fxn)(hWnd, wMsg, wParam, lParam));
    }

    return(DefWindowProc(hWnd, wMsg, wParam, lParam));
}

//
// DoCommand -- processes WM_COMMAND messages for frame window.
// Decodes the menubar item with the menuitems[] array, then
// runs the corresponding function to process the command.
// If no match found, passes message to Windows DefWindowProc().
//
LONG DoCommand(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    INT i;

    for(i = 0; i < dim(menuitems); i++)
    {
        if(wParam == menuitems[i].Code)
            return((*menuitems[i].Fxn)(hWnd, wMsg, wParam, lParam));
    }

    return(DefWindowProc(hWnd, wMsg, wParam, lParam));
}

//
// DoDestroy -- processes the WM_DESTROY message for frame window by
// posting a WM_QUIT message to the same window, forcing WinMain
// to fall out of the event loop.
//
LONG DoDestroy(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    PostQuitMessage(0);
    return(FALSE);
}

//
// DoClose -- processes a WM_CLOSE message for frame window by
// saving the current window position, size, filename, and file
// offset, then forcing a WM_DESTROY message.
//
LONG DoClose(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    UpdateFrameProfile();                   // save window information
    DestroyWindow(hWnd);                    // then close down the app
    return(FALSE);
}

//
// DoVScroll -- process various WM_VSCROLL messages for frame window.
// The user can generate these messages by dragging thumb, clicking
// in scrollbar, or clicking arrows at both ends of scrollbar.
//
LONG DoVScroll(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    RECT rect;

    switch(LOWORD(wParam))                  // LOWORD is vital for Win32
    {
        case SB_TOP:                        // go to top of file if
            if(ViewPtr)                     // we aren't there already
            {
                SetFilePosition(0);
                Repaint();
            }
            break;

        case SB_BOTTOM:                     // go to bottom of file if
            SetFilePosition(FileSize);      // we aren't there already
            Repaint();
            break;

        case SB_LINEUP:                     // scroll up by one line if
            if(ViewPtr)                     // we aren't already at top
            {
                SetFilePosition(ViewPtr - BPL);
                ScrollWindow(hWnd, 0, CharY, NULL, NULL);
                UpdateWindow(hWnd);
            }
            break;

        case SB_LINEDOWN:                   // scroll down by one line if
            if(ViewPtr < TopAddr)           // we aren't already at bottom
            {
                SetFilePosition(ViewPtr + BPL);
                ScrollWindow(hWnd, 0, -CharY, NULL, NULL);
                GetClientRect(hWnd, &rect);
                rect.top = max(0, (LPP-1) * CharY);
                InvalidateRect(hWnd, &rect, TRUE);
                UpdateWindow(hWnd);
            }
            break;

        case SB_PAGEUP:                     // scroll up by one page
            SetFilePosition(ViewPtr - BPP);
            Repaint();
            break;

        case SB_PAGEDOWN:                   // scroll down by one page
            SetFilePosition(ViewPtr + BPP);
            Repaint();
            break;

        case SB_THUMBPOSITION:              // reposition file by thumb
            SetFilePosition(ThumbInc * (LONG) THUMBPOS);
            Repaint();
            break;

        case SB_THUMBTRACK:                 // track drag of thumb
            ThumbTrack(THUMBPOS);
            break;
    }

    return(FALSE);
}

//
// DoPaint -- process WM_PAINT message for frame window by
// painting hex and ASCII data dump for file offsets that fall
// within the window.  We make no attempt to optimize this for
// painting regions that are smaller than the window.
//
LONG DoPaint(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    HDC hdc;                                // scratch device context
    PAINTSTRUCT ps;                         // scratch paint structure
    INT i;

    hdc = BeginPaint(hWnd, &ps);            // get device context
    SelectObject(hdc, hFont);               // realize nonprop. font

    if(hFile != -1)                         // paint all lines that
    {                                       // fall within the window
        for(i = 0; i < LPP; i++)
```

HEXVIEW.C

3 of 4

```
            DisplayLine(hdc, i);
    }

    EndPaint(hWnd, &ps);                    // release device context
    return(FALSE);
}

//
// DoChildPaint -- process WM_PAINT message for child window.
// These occur during thumb drag; we put up a tiny window that
// displays the file offset corresponding to the thumb position.
//
LONG DoChildPaint(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    HDC hdc;
    PAINTSTRUCT ps;
    RECT rect;
    char buff[256];

    hdc = BeginPaint(hWnd, &ps);            // get device context
    GetClientRect(hWnd, &rect);             // get client area dimensions
    SelectObject(hdc, hFont);               // select nonproportional font
    wsprintf(buff, "%08lX", FileIndex);     // format file index from thumb
    DrawText(hdc, buff, -1,                 // paint index into window
        &rect, DT_CENTER | DT_VCENTER | DT_SINGLELINE);
    EndPaint(hWnd, &ps);                    // release device context
    return(FALSE);
}

//
// DoSize -- process WM_SIZE message for frame window by calculating
// the number of lines per page and bytes per page for new window size.
// Also reposition the display if we were already at end of file and
// the window grew.
//
LONG DoSize(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    LPP = HIWORD(lParam) / CharY;           // calc lines per page
    BPP = LPP * BPL;                        // calc bytes per page
    ConfigDisplay();                        // calc display parameters
    if(ViewPtr > TopAddr)                   // make sure window refilled
        SetFilePosition(TopAddr);
    return(FALSE);
}

//
// DoMenuOpen -- process File-Open command from menu bar. All
// the hard work is done by the OpenFile common dialog.
//
LONG DoMenuOpen(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    OPENFILENAME ofn;                       // used by common dialogs

    szFileName[0]  = '\0';                  // init filename buffer

    ofn.lStructSize = sizeof(OPENFILENAME); // length of structure
    ofn.hwndOwner = hWnd;                   // handle for owner window
    ofn.lpstrFilter = szFilter[0];          // address of filter list
    ofn.lpstrCustomFilter = NULL;           // custom filter buffer address
    ofn.nFilterIndex = 1;                   // pick default filter
    ofn.lpstrFile = szFileName;             // buffer for path+filename
    ofn.nMaxFile = EXENAMESIZE;             // length of buffer
    ofn.lpstrFileTitle = NULL;              // buffer for filename only
    ofn.lpstrInitialDir = NULL;             // initial directory for dialog
    ofn.lpstrTitle = NULL;                  // title for dialog box
    ofn.Flags = OFN_PATHMUSTEXIST | OFN_FILEMUSTEXIST;
    ofn.lpstrDefExt = NULL;                 // default extension

    if(GetOpenFileName(&ofn))               // display open dialog
    {
        OpenDataFile();                     // open file for viewing
        ReadDataFile();                     // initialize data buffer
        Repaint();                          // force display of data
    }

    return(FALSE);
}

//
// DoMenuExit -- process File-Exit command from menu bar.  This
// is simply handled by sending a WM_CLOSE message as though Close
// had been picked on the System menu to shut down the app.
//
LONG DoMenuExit(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    SendMessage (hWnd, WM_CLOSE, 0, 0L);
    return(FALSE);
}

//
// DoMenuAbout -- process File-About command from menu bar. We
// allocate a thunk for the dialog callback routine, display the
// dialog, then release the thunk after the dialog is dismissed.
//
LONG DoMenuAbout(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    WNDPROC lpProcAbout;

    lpProcAbout = MakeProcInstance((WNDPROC)AboutDlgProc, hInst);
    DialogBox(hInst, "AboutBox", hWnd, lpProcAbout);
    FreeProcInstance(lpProcAbout);
    return(FALSE);
}

//
// AboutDlgProc -- This is the callback routine for the About... dialog.
//
BOOL FAR APIENTRY AboutDlgProc (HWND hwnd, UINT msg, UINT wParam, LONG lParam)
{
    if((msg==WM_COMMAND) && (wParam==IDOK)) // dismiss dialog if OK
        EndDialog(hwnd, 0);                 // button was clicked,
    return(FALSE);                          // otherwise do nothing
}

//
// SetWindowCaption -- concatenate the filename with the application
// name, then update the frame window's title bar.
//
VOID SetWindowCaption(char * szFilename)
{
    char szTemp[EXENAMESIZE+1];

    strcpy(szTemp, szAppName);              // get application name
    strcat(szTemp, " - ");                  // add separator
    strcat(szTemp, szFileName);             // add filename
    SetWindowText(hFrame, szTemp);          // put result into title bar
}

//
// Repaint -- force refresh of formatted output in main window. This
// gets called at various points when the file position is changed.
//
VOID Repaint(VOID)
{
    InvalidateRect(hFrame, NULL, TRUE);     // repaint entire window
}

//
// SetFilePosition - called during processing of various vertical scrollbar
// messages to set ViewPtr to a valid value.
//
VOID SetFilePosition(LONG NewViewPtr)
{
    if(hChild)                              // destroy the child window
    {                                       // if it happens to be active
        DestroyWindow(hChild);
        hChild = (HWND) 0;
    }

    if(hFile == -1)                         // bail out if no file open
        return;

    ViewPtr = NewViewPtr & 0x0FFFFFF0;      // offset is multiple of 16

    if(ViewPtr > TopAddr)                   // enforce legal file offset
        ViewPtr = TopAddr;
    if(ViewPtr < 0)
        ViewPtr = 0;

    SetScrollPos(hFrame, SB_VERT,           // set thumb on scrollbar
        ViewPtr/ThumbInc, TRUE);
}

//
// ConfigDisplay -- Configure various display parameters and scrollbar
// range according to current window size and file size.
//
VOID ConfigDisplay(VOID)
{
    // calc address and line number of first line, last page
    TopAddr = max(((FileSize + 15) & 0x0FFFFFF0) - BPP, 0);
    TopLine = TopAddr / BPL;

    // calculate bytes per scrollbar increment
    ThumbInc = BPL * ((TopLine/32767) + 1);

    // configure vertical scrollbar or make it disappear if not needed
    if(BPP >= FileSize)                     // file fits within window
    {
        SetScrollRange(hFrame, SB_VERT, 0, 0, FALSE);
        SetScrollPos(hFrame, SB_VERT, 0, TRUE);
        if(ViewPtr) SetFilePosition(0);
    }
    else                                    // file is too big for window
    {
        SetScrollRange(hFrame, SB_VERT, 0, ((TopLine*BPL)/ThumbInc), FALSE);
        SetScrollPos(hFrame, SB_VERT, ViewPtr/ThumbInc, TRUE);
    }
}

//
// ThumbTrack() - called during processing of WM_VSCROLL message to track
// thumb position, also forcing update of file offset in child window.
//
VOID ThumbTrack(INT pos)
{
```

HEXVIEW.C

4 of 4

```
    RECT rect;
    INT SizeX = CharX * 10;                 // calc size of child window
    INT SizeY = CharY * 2;

    if(!hChild)                             // create child window
    {
        GetClientRect(hFrame, &rect);       // get client area of frame

        hChild = CreateWindow(              // create child window
            szChildClass,                   // window class name
            NULL,                           // text for title bar
            WS_CHILD | WS_BORDER | WS_VISIBLE,  // window style
            (rect.right-SizeX) / 2,         // x position
            (rect.bottom-SizeY) / 2,        // y position
            SizeX,                          // window width
            SizeY,                          // window height
            hFrame,                         // frame is parent window
            (HMENU) 1,                      // child window ID
            hInst,                          // window owner module
            NULL);                          // unused pointer
    }

    // calculate file location to display, force repaint of child window
    FileIndex = min((LONG) pos * (LONG) ThumbInc, TopAddr);
    InvalidateRect(hChild, NULL, TRUE);
}

//
// DisplayLine -- format and display a single line of hex/ASCII dump
// using the device context and relative window line number supplied
// by the caller.  The file data offset is calculated from the
// relative window line and the offset corresponding to the line
// currently at the top of the window.
//
VOID DisplayLine(HDC hdc, INT line)
{
    INT i;
    char buff[256], c, *p;
    LONG x;

    if((ViewPtr + (LONG) line*BPL) < FileSize)
    {
        // format file offset as 8-digit hex number
        p = buff + wsprintf(buff, "%08lX ", ViewPtr + (LONG) line*BPL);

        // format this 16-bytes as hex data
        for(i = 0; i < BPL; i++)
        {
            x = ViewPtr + (LONG)(line * BPL) + i;

            if(x < FileSize)
                p += wsprintf(p, "%02X ", GetByte(x) & 0x0ff);
            else
                p += wsprintf(p, "   ");
        }

        // format same 16-bytes as ASCII, using "." for control characters
        for(i = 0; i < BPL; i++)
        {
            x = ViewPtr + (LONG)(line * BPL) + i;

            if(x < FileSize)
            {
                c = GetByte(x);
                if(c < 0x20)
                    c = '.';
            }
            else c = ' ';
            *p++ = c;
        }

        // append a null byte, then paint the formatted data
        *p = '\0';
        TextOut(hdc, 0, line*CharY, buff, strlen(buff));
    }
}

//
// GetByte -- retrieve data byte from file I/O buffer, performing
// disk I/O as necessary.  This is a simple-minded virtual memory
// scheme when file is larger than buffer.
//
char GetByte(LONG addr)
{
    if(!((addr >= FilePtr) && (addr < (FilePtr + BUFSIZE))))
    {
        FilePtr = (addr - (BUFSIZE/2)) & 0x0FFFFFF0;
        if(FilePtr < 0) FilePtr = 0;
        ReadDataFile();
    }

    return(lpBuff[addr - FilePtr]);
}

//
// OpenDataFile -- open specified file for viewing, save handle.  The
// filename was previously placed in szFileName by OpenFile common dialog.
//
VOID OpenDataFile(VOID)
{
    if(hFile != -1)                         // close previous file if any
        _lclose(hFile);

    hFile = _lopen(szFileName, OF_READ);    // try and open the new file

    if(hFile == -1)                         // bail out if no such file
    {
        MessageBox(hFrame, "Can't open file!", szAppName, MB_ICONSTOP|MB_OK);
        return;
    }

    FileSize = _llseek(hFile, 0, 2);        // get size of file
    ViewPtr = 0;                            // reset window address pointer
    FilePtr = 0;                            // reset file i/o pointer
    ConfigDisplay();                        // calc display parameters
    SetWindowCaption(szFileName);           // update title bar
}

//
// ReadDataFile -- read data from specified file to I/O buffer, using
// current file position set by caller.
//
VOID ReadDataFile(VOID)
{
    if(hFile == -1)                         // bail out if no file open
        return;
    _llseek(hFile, FilePtr, 0);             // position file pointer
    _lread(hFile, lpBuff, BUFSIZE);         // read the file
}

//
// UpdateFrameProfile() --  Update frame window profile in INI file
// (if Win 3.x) or registration database (if NT/WIN32) by saving the
// current window position, size, name of file being displayed, file offset.
//
VOID UpdateFrameProfile(VOID)
{
    RECT rect;
    char temp[20];

    if(IsIconic(hFrame) || IsZoomed(hFrame)) return;

    GetWindowRect(hFrame, &rect);           // get position of frame window

    wsprintf(temp,"%d", rect.left);
    WritePrivateProfileString("Frame", "xul", temp, "HexView.ini");

    wsprintf(temp,"%d", rect.top);
    WritePrivateProfileString("Frame", "yul", temp, "HexView.ini");

    wsprintf(temp,"%d", rect.right);
    WritePrivateProfileString("Frame", "xlr", temp, "HexView.ini");

    wsprintf(temp,"%d", rect.bottom);
    WritePrivateProfileString("Frame", "ylr", temp, "HexView.ini");

    WritePrivateProfileString("File", "filename", szFileName, "HexView.ini");

    wsprintf(temp,"%ld", ViewPtr);
    WritePrivateProfileString("File", "fileptr", temp, "HexView.ini");
}
```

variable declarations and at the function prototypes and declarations, you'll see that they almost uniformly use TYPEDEFs in preference to real or native data types. You'll also notice that a few of the TYPEDEFs are strategically remapped for the two different execution environments in the HEXVIEW.H header file (the remainder are handled in WINDOWS.H, which will handle all of them in the final versions of the development systems).

Second, we must take into account the different calling conventions in WIN16 and WIN32. WIN16 applications use a mixture of the CDECL calling convention (parameters are pushed right to left, and the caller clears the stack) and the PASCAL calling convention (parameters are pushed left to right, and the called item clears the stack). In general, routines that are entered directly from Windows (message and dialog callbacks) are declared as FAR PASCAL, while the

HEXVIEW.H
Complete Listing

```
//
// HexView.H -- Header File for HexView.C
//

#if !defined(WIN32)

  #define WIN16     TRUE
  #define WIN31

  #define INT       int
  #define UINT      WORD
  #define APIENTRY  PASCAL
  #define WNDPROC   FARPROC
  #define THUMBPOS  LOWORD(lParam)

#else

  #define WIN16     FALSE
  #define THUMBPOS  HIWORD(wParam)

#endif

struct decodeWord {                          // structure associates
    UINT Code;                               // messages or menu IDs
    LONG (*Fxn)(HWND, UINT, UINT, LONG); }; // with a function

#define     IDM_OPEN     100
#define     IDM_EXIT     101
#define     IDM_ABOUT    102
```

```
// Function prototypes
INT APIENTRY WinMain(HANDLE, HANDLE, LPSTR, INT);
BOOL InitApp(HANDLE);
BOOL InitInstance(HANDLE, INT);
BOOL TermInstance(HANDLE);
LONG FAR APIENTRY FrameWndProc(HWND, UINT, UINT, LONG);
LONG FAR APIENTRY ChildWndProc(HWND, UINT, UINT, LONG);
BOOL FAR APIENTRY AboutDlgProc (HWND, UINT, UINT, LONG);
LONG DoDestroy(HWND, UINT, UINT, LONG);
LONG DoClose(HWND, UINT, UINT, LONG);
LONG DoPaint(HWND, UINT, UINT, LONG);
LONG DoChildPaint(HWND, UINT, UINT, LONG);
LONG DoSize(HWND, UINT, UINT, LONG);
LONG DoCommand(HWND, UINT, UINT, LONG);
LONG DoVScroll(HWND, UINT, UINT, LONG);
LONG DoMenuOpen(HWND, UINT, UINT, LONG);
LONG DoMenuExit(HWND, UINT, UINT, LONG);
LONG DoMenuAbout(HWND, UINT, UINT, LONG);
VOID OpenDataFile(VOID);
VOID ReadDataFile(VOID);
VOID SetWindowCaption(char *);
VOID Repaint(VOID);
VOID SetFilePosition(LONG);
VOID ConfigDisplay(VOID);
VOID ThumbTrack(INT);
VOID DisplayLine(HDC hdc, INT line);
char GetByte(long);
VOID UpdateFrameProfile(VOID);
```

Figure 5: **This is the C-language header file for the HEXVIEW utility.**

procedures that are called only from within the application itself are declared as NEAR or FAR CDECL.

In WIN32, on the other hand, all calls are NEAR because of the flat memory model, and by default the compiler uses hybrid calling convention named STDCALL where parameters are pushed right to left but the callee clears the stack. These differences are disguised in the source code by again treating FAR as a NOOP for 32-bit code and by hiding the actual calling convention beneath #defines such as APIENTRY, CALLBACK, and WNDPROC.

Amazingly, if we've been careful not to use any of the API functions in our program that are not completely symmetric between WIN16 and WIN32, the masking of data types and calling conventions handles about 90 percent of the portability considerations for WIN16 and WIN32.

Message packets and message-parameter packing constitute most of the rest of our worries. WIN16 message packets have three 16-bit components (hWnd, wMsg, and wParam) and one 32-bit component (lParam). WIN32 message packets have four components with the same names, but all are 32-bit. Just as with the API function parameters, this difference is concealed behind TYPEDEFs (UINT and LONG).

The issue of parameter packing is a little more tricky. We first look at all the messages processed by routines intrinsic to our program (these are nicely summarized in the tables frameMsg[] and childMsgs[]) and compare them to the list of messages in Figure 2 that were redefined for WIN32. The one message that turns up as a potential problem for HEXVIEW is WM_VSCROLL. Turning now to the programmers' references for WIN16 and WIN32, we find the following information:

- WIN16 WM_VSCROLL: wParam = scroll event type (SB_LINEUP, SB_PAGEUP, and so forth); lParam = window handle in high word, thumb position in low word (if applicable).
- WIN32 WM_VSCROLL: wParam = scroll event type (SB_LINEUP, SB_PAGEUP, and so forth) in low word, thumb position in high word (if applicable); lParam = window handle.

Some of these differences are easily handled with the conditional #defines of the general form:

```
#if defined(WIN32)
#define THUMBPOS HIWORD(wParam)
#else
#define THUMBPOS LOWORD(lParam)
#endif
```

HEXVIEW.RC
Complete Listing

```
// Resource script for HEXVIEW.C
#include "windows.h"
#include "hexview.h"
HexViewIcon ICON hexview.ico
HexViewMenu MENU
BEGIN
    POPUP           "&File"
    BEGIN
        MENUITEM    "&Open",     IDM_OPEN
        MENUITEM    "E&xit",     IDM_EXIT
        MENUITEM    SEPARATOR
        MENUITEM    "A&bout",    IDM_ABOUT
    END
END
AboutBox DIALOG 22, 17, 126, 53
CAPTION "About HexView..."
STYLE DS_MODALFRAME | WS_CAPTION | WS_SYSMENU
BEGIN
    ICON "HexViewIcon", -1, 7, 8, 16, 16, WS_CHILD | WS_VISIBLE
    CTEXT "Hex File Viewer 1.0", -1, 33, 6, 84, 8
    CTEXT "\251 1992 Petit Mal Software", -1, 32, 16, 87, 9
    CONTROL "OK", IDOK, "BUTTON", WS_GROUP, 47, 35, 32, 14
END
```

Figure 6: **This is the resource script for the HEXVIEW utility.**

then referencing the symbol THUMBPOS in the source code wherever the thumb position is needed. (Microsoft recommends use of its "message cracker" macros instead, but I found them ugly and clumsy, and gave up on them almost immediately.) The remainder of environmental differences between WM_VSCROLL messages are accounted for by using LOWORD (wParam) whenever the scroll event type is examined; the LOWORD does no harm in WIN16 and ensures the proper isolation of the scroll event type in WIN32.

One very important divergence between the WIN16 and WIN32 environments, which does not show up in our HEXVIEW code at all, is the handling of the information that is stored and retrieved by the Windows functions WritePrivateProfileString, GetPrivateProfileString, and GetPrivateProfileInt. In Windows 3.*x*, this information is stored in a plain ASCII text file named (in this case) HEXVIEW.INI in the C:\WINDOWS directory. Under NT, the information is stored in an indexed binary database on a per-user basis, protected by access control lists.

As it stands, HEXVIEW is a WIN32S application and takes no special advantage of NT's capabilities when running as a 32-bit program. NT-specific enhancements can easily be added via conditional compilation, but these will tend to make the source code more confusing, so the potential performance gains must be weighed against the damage to maintainability. Our first inclination might be simply to make the file I/O buffer a lot bigger under WIN32 (since the flat memory model makes access to multimegabyte data objects very efficient). But this would probably slow the program down, rather than speed it up. The allocation of a huge buffer will stir up a lot of activity in the system's virtual memory manager, and in the case of very large files HEXVIEW will waste time reading data that is never needed. A much better solution under NT would use its mapped file functions, which essentially associate offsets within a disk file with a range of memory addresses, and would shift the entire burden of file I/O onto the system's paged memory manager.

HEXVIEW.DEF
Complete Listing

```
NAME          HexView

DESCRIPTION   'HexView - Windows Hex Viewer'

EXETYPE       WINDOWS

STUB          'WINSTUB.EXE'

CODE          PRELOAD MOVEABLE
DATA          PRELOAD MOVEABLE MULTIPLE

HEAPSIZE      32768
STACKSIZE     8192

EXPORTS
  FrameWndProc
  AboutDlgProc
  ChildWndProc
```

Figure 7: **This is the module definition file for the HEXVIEW utility.**

THE IN-BOX Please send your questions, comments, and suggestions to me at any of these electronic-mail addresses:
PC MagNet: 72241,52
MCI Mail: rduncan
BIX: rduncan
Internet: duncan@csmcmvax.bitnet □